Extending Summation Precision for Network Reduction Operations

George Michelogiannakis, Xiaoye S. Li, David H. Bailey, John Shalf
Lawrence Berkeley National Laboratory, 1 Cyclotron Road, Berkeley, CA 94720
Email: {mihelog,xsli,dhbailey,jshalf}@lbl.gov

***Abstract*—Double precision summation is at the core of numerous important algorithms such as Newton-Krylov methods and other operations involving inner products, but the effectiveness of summation is limited by the accumulation of rounding errors, which are an increasing problem with the scaling of modern HPC systems and data sets. To reduce the impact of precision loss, researchers have proposed increased- and arbitrary-precision libraries that provide reproducible error or even bounded error accumulation for large sums, but do not guarantee an exact result. Such libraries can also increase computation time significantly. We propose big integer (BigInt) expansions of double precision variables that enable arbitrarily large summations *without error* and provide exact and reproducible results. This is feasible with performance comparable to that of double-precision floating point summation, by the inclusion of simple and inexpensive logic into modern NICs to accelerate performance on large-scale systems.**

I. Introduction

Technology scaling and architecture innovations in recent years have led to larger datasets and extreme parallelism for high performance computing (HPC) systems. It is projected that parallelism will increase at an exponential pace in order to meet the community's computational and power efficiency goals [1]. This translates to operations, such as summations, with a larger number of operands that are also distributed over an increasing number of nodes. The sheer scale of these systems threatens the numerical stability of core algorithms due to the accumulation of errors for ubiquitous global arithmetic operations such as summations [2].

At the present time, 64-bit IEEE floating-point arithmetic is sufficiently accurate for most scientific applications. However, for a rapidly growing body of important scientific applications, higher numeric precision is required. Among those applications are computations with large, relatively ill-conditioned linear systems, multi-processor large-scale simulations, atmospheric models, supernova simulations, Coulomb n-body atomic system studies, mathematical physics analyses, studies in dynamical systems, and computations for experimental mathematics. Recent papers discuss numerous such examples [3], [4], [5], [6], [7].

Perhaps the most common instance where extra precision is needed is large summations, particularly when it involves both positive and negative terms, so that catastrophic cancellation can occur. Even if all terms are positive, round-off error when millions or more terms are summed can significantly reduce the accuracy of the result, especially if the summation involves large and small numbers. This is important even for simple widely-used operations based on summations, such as matrix multiplications and dot products.

As an example of a real-world application where accurate summation is important, Y. He and C. Ding analyzed anomalous behavior in large atmospheric model codes, wherein significantly different results were obtained on different systems, or even the same system with a different numbers of processors [7]. As a result, code maintenance was an issue, since it was often difficult to ensure that a "bug" was not introduced when porting the code to another system. While a certain degree of non-reproducibility is to be expected since weather and climate calculations are fundamentally chaotic, these These researchers found that almost all of the numerical variations were eliminated when two key inner summations were converted to use double-double (DD) arithmetic (approximately 31 decimal digits) [7]. Another concrete example is complex dynamical systems, where precision loss has been quoted as the main limitation [4].

Currently, researchers generally presume that nothing can be done to ameliorate such problems and are often concerned with bounding errors from precision loss to acceptable levels [8], [9], [10], [11], [12], [13]. A few have tried higher-precision arithmetic using software facilities, with relatively good success, although computational time for this software is often an issue [14], [15]. In the atmospheric model case mentioned above, researchers employed DD arithmetic in the summations, which uses two double-precision variables to extend precision [14], and found that almost all numerical variations were eliminated. Others have even resorted to quad-double arithmetic [3], [14].

In modern HPC systems with large datasets, summations and other operations are often performed on operands distributed across the network [16], [1]. Currently none of the high-precision software libraries support parallel summations. Balanced-tree summations and other sorting or recursion algorithms that reduce round-off error accumulation [10], [15], [9], [12] are only efficient within nodes because they would incur an impractical amount of system-wide data movement for distributed operations. This leaves distributed operations prone to precision loss.

This work was supported by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE- AC02-05CH11231.

There are two natural ways to implement distributed summations with provably minimum data movement. One way is for nodes to send their partial sum (a single variable) to a single node to perform the summation and generate a single result. This only occupies a single node's processor but completion time grows linearly with the number of operands and can also create network hotspots [17]. The other approach (used in most MPI implementations) performs the summation in a tree fashion, where each node adds a number of operands and passes the partial sum (also a single variable) to the next level of the tree. This approach requires logarithmic time to complete, but the latencies of each level of the tree greatly impact performance at scale as it incurs multiple communication delays between network interface cards (NICs) and local processors. Software implementations incur potentially long context switch penalties just to have processors at each node add a few numbers together, and may also cause these intermediate processors to wake up from potentially deep sleep state just to perform an infinitesimal amount of work.

Earlier work proposes NIC-based reductions in Myrinet networks [18] by modifying the drivers (and therefore using the host processor's processing power), and showed that NIC-based reduction can yield performance gains even with as few as eight nodes [19]. Further work evaluated conducting reduction operations exclusively in NICs and concluded that doing so increases efficiency, scalability and reduces execution time compared to performing the computations in processors [20]. This is made possible by using the programmable logic in modern NICs [20]. However, this work only applied this method for double-precision variables due to the complex data structures and computation demands of arbitrary-precision libraries and the limited processing power of programmable logic in NICs. Even with double-precision variables, the low computational power of programmable logic can pose a significant performance issue [20]. The alternative, adding complex dedicated hardware to NICs for increased or arbitrary precision computations, increases design risk and complexity.

In this paper, we make the case that using a very wide integer to provide an uncompressed and exact encoding of the complete numerical space that can be represented by a double-precision floating point number, which we call big integer (BigInt), is well-suited for distributed operations. We demonstrate that BigInts incur no precision loss and can be productive and efficient to use in distributed summations, such as reduction operations, without increasing communication latency compared to double-precision variables. In fact, this approach offers the promise of actually accelerating system-wide summations used by inner products for methods such as Newton-Krylov, because integer adders can be easily incorporated in NICs at speeds comparable to modern floating point units (FPUs), to avoid waking up and context switching processors. Therefore, we are able to move computation from the processor to the NIC without any precision or performance loss. This was previously possible only with double-precision variables due to the complex internal structures of arbitrary-precision libraries. Even though the size of a BigInt variable is 2101 bits—33× larger than a 64-bit double-precision variable—we observe insignificant loss in communication delay due to the fixed latency costs and packet overhead bytes in modern large-scale networks [21]. Therefore, BigInts readily apply to network operations and can be combined with past work on local-node computations that uses sorting and recursion or alternative wide fixed-point representations with dedicated hardware support [22], [23], [15], in order to provide reproducible system-wide operations with no precision loss.

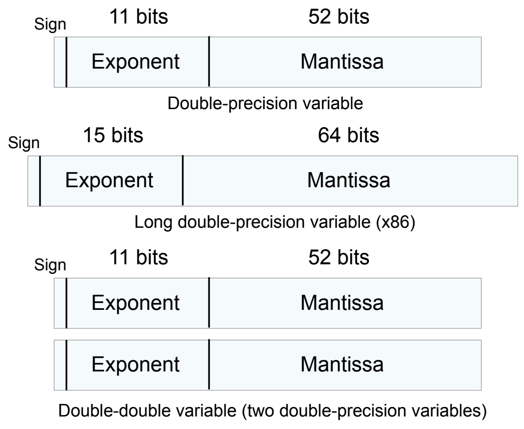


Figure 1: Double, long double and double-double variables.

II. Background and Related Work

The format of double-precision variables is set by the IEEE 754 standard [24] and is shown in Figure 1. For double-precision variables, the total number of bits is 64, including the most significant bit which is the sign. The mantissa includes a silent bit at its most significant (53rd) position, which is 1 if the exponent is non-zero. Double-precision variables with a zero exponent are considered denormalized and are used to fill the gaps in the numbers that double-precision variables can represent close to zero. A denormalized double-precision variable encodes the number:

$$DenormalizedValue = (-1)^{sign} \times 0.Mantissa \times 2^{-1022}$$

If the exponent is non-zero, the number is considered normalized and a bias of -1023 is applied to the exponent value. Therefore, an exponent value of 50 represents an actual exponent of $50 - 1023 = -973$. This way, double-precision variables can represent numbers with negative exponents. The decimal value of a normalized double is:

$$NormalizedValue = (-1)^{sign} \times [1 + Mant_{51}(\frac{1}{2}) + Mant_{50}(\frac{1}{4}) + ... + Mant_0(1/2^{52})] \times 2^{Exp-1023}$$

Normalized doubles encode numbers in the range:

$$2^{-1022} \leq |Value| \leq (2 - 2^{-52}) \times 2^{1023}$$

However, because of the limited mantissa bits, double-precision variables cannot represent all numbers in the above range [4]. In addition, operating on numbers that have significantly different exponent values causes bits to be dropped. For example, adding the numbers 2^N and 2^M where $N >= M+53$ causes the latter number to be dropped because the 53 mantissa bits are not enough to retain both numbers. Even if the smaller number is not small enough to be fully discarded, it may have to be partly discarded for the same reason, or the result may require more than 53 mantissa bits [4]. Such precision loss is hard to predict in many applications and can appear with just two operands. In a large summation this error accumulates in every sum of two numbers and therefore in a summation with millions or billions of operands the error may rise to unacceptable levels [3], [7], [5], [6].

Double-precision variables are fully supported in hardware with dedicated FPUs in modern processors [25]. They also include special values to represent infinity and a result that is not a number (NaN), as well as exceptions to set variables to those values when appropriate [24], [26]. Modern architectures include support for "long double" variables. In the IEEE 754 standard [24], this is referred to as double-extended precision for which the byte width is 10 and translates to 19 or 20 decimal digits. The implementation of these variables depends on the programming environment and system architecture. In an x86 architecture long doubles are 80-bits long, with 64-bit mantissas and 15-bit exponents [27]. x86 processors include dedicated registers with this format, but the extra precision is lost once a long double variable is written back to memory.

In the past 10 years or so, high-precision software packages have been produced which typically utilize custom datatypes and operator overloading to facilitate conversion. One example is the QD package [14] which provides DD variables, where each 16-byte variable is the unevaluated sum of two 8-byte doubles of which the first consists of the "leading" digits and the second the "trailing digits" of the format's value. That is, the first double contains the best 64-bit double-precision approximation to the answer, and the second double contains the difference between the exact answer and the first double. Similarly, a quad-double number is an unevaluated sum of four IEEE doubles. However, while such packages offer easy-to-use high-level interfaces, they still offer a limited amount of precision.

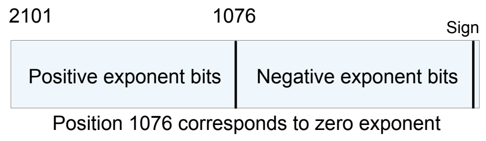


Figure 2: A BigInt consists of 2101 bits. Bit positions correspond to an exponent after a bias of 1076. Positions right of 1076 correspond to negative exponents. For example, position 1000 corresponds to exponent $1000 - 1076 = -76$.

For arbitrary precision, packages such as ARPREC [28] and GNU GMP [29] are available. Other packages offer guarantees on the order of operations and rounding (since there is no standard) [30]. Inevitably, these packages sacrifice speed and memory due to large arrays used in the internal representations and lack of hardware support. Formats with variable-length mantissas have also been proposed, such as IEEE 854 [31]. Such formats are similar to double-precision variables because they use mantissa and exponent fields, but they also introduce additional complexity for operations and exception handling that make hardware support costly. Finally, wide fixed-point representations similar to BigInts have been used in hardware accumulators for local-node summations [22], [23], [15]. BigInts extend this work by applying this concept on distributed summations.

We found that existing software solutions are usually slow for most physical modeling and simulation codes, and the precision provided by DDs or quad-doubles are insufficient, especially for much larger simulations to come. A DD summation requires approximately 20 operations, all performed in processor registers, which results in an approximately 2× to 5× slowdown compared to double-precision variables [14]. This slowdown becomes worse when using arbitrary-precision packages. As an example, ARPREC is approximately 2.5× to 4.5× slower than DD multiplication on a Sun Ultra 167 MHz processor [28], [14]. Alternative algorithms use sorting, recursion, or other computation or data movement overhead to provide exact results or with tight error bounds, as well as reproducibility [15], [10], [9]. Due to their overhead, such algorithms are only practical within nodes. Because of the performance tradeoff, some researchers settle with bounding or predicting the computation error [8], [9], [10], [11], [12], [13].

III. Description

A. Big Integer Variables

Figure 2 illustrates the format of a BigInt variable. Essentially, BigInt variables remove the exponent field and expand the mantissa such that each bit's location corresponds to an exponent value. To represent the same number space as the IEEE double-precision format, BigInt variables need to be 2101 bits since the 11-bit exponent field's value ranges from 0 to 2048, and the exponent defines the value of the most

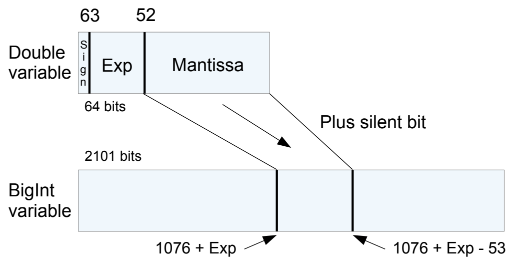


Figure 3: A double-precision variable maps to a BigInt by inserting the most significant bit of the mantissa (the silent bit if set) in the position specified by the exponent. For example, an exponent of 1000 encodes an actual exponent of -23 after applying the -1023 bias. Therefore, the silent bit is inserted in position $1076 - 23 = 1053$. The rest of the mantissa is inserted in adjacent less significant positions.

significant (silent) mantissa bit. Thus, the lowest exponent value BigInt needs to represent is the lowest value of the exponent field (-1022) minus the number of mantissa bits (53), for a total of -1075. BigInt variables also have a sign bit which we place in the least significant position.

Like integers, BigInts are an uncompressed representation and therefore can exactly encode the full range of real numbers in their number space. In other words, there is no gap in the range of numbers that are expressible by BigInts. In addition, BigInts do not introduce any precision loss. Therefore, they are well-suited for very large summations.

Figure 3 shows how a normalized double-precision variable maps to a BigInt. To convert from a denormalized double, the most significant bit of the double's mantissa is inserted in position 53 (corresponding to exponent value -1022). This does not overwrite the sign bit of BigInt because denormalized floats do not have a silent bit.

Converting the summation result from a BigInt back to a double, long double, or DD variable inevitably causes the least significant asserted bits to be dropped if the distance between the leftmost and rightmost asserted bits in the BigInt is more than the length of the mantissa of the variable the BigInt is converted to. For example, when converting to a double-precision variable a BigInt with asserted bits at locations 100 and 10, the least significant bit will inevitably be dropped because $100 - 10 > 53$. If this precision loss is of concern to the application, programmers can retain BigInt or use another wide fixed-point for local-node summation similar to past work [22], [23], [15], or use any of the higher-precision formats or libraries such as recursion and sorting methods, ARPEC, QD, and GMP [28], [14], [29], [15], [10], [9]. It is key to realize that the conversion back to lower-precision formats discussed above is performed only for the final result or for local node processing, if at all. The final result is guaranteed to be accurate as long as all arithmetic is performed with BigInt precision. Performing the entire operation with precision less than BigInt may produce an entirely different result due to precision loss in every summation of two numbers, as discussed in Section II; this effect accumulates over the course of large summations. This precision loss can result in the most significant bits of the result, the bits well within the range of even double-precision variables, to differ compared to the result produced by BigInt because BigInt was able to correctly and exactly capture the contributions of all operands without losses.

The simplicity of BigInts enables fast and accurate numerics for software implementations and requires only simple logic for hardware implementations for large summations [22], [23]. Very little additional memory is required because only partial sums are projected to BigInt, instead of all operands, and only over the network for distributed operations. BigInt variables can be treated as long integers for addition, multiplication and division, even though they represent floating point numbers. Adding two BigInts can be accomplished with simple and inexpensive integer adders in a sequential manner; an adder of N bits length requires $\frac{2101}{N}$ cycles. Therefore, in order to match the latency of a highly-optimized FPU in modern Intel processors which has a 5-cycle latency [27], the adder width needs to be 421 bits. This makes adding hardware support for BigInts feasible and more likely to operate at comparable speeds to the node processors, in contrast to more complex libraries that also offer arbitrary precision.

BigInts of all 0s but with a sign bit of 1 represents "not a real number" (NaN). A value of all 1s refers to infinity (Inf), used for numbers outside of the representable number space. In this case, the sign bit distinguishes between positive and negative infinity, similar to IEEE 754 [24], [26]. Setting a BigInt to infinity or NaN is done by the hardware logic through simple checks without the need for exceptions, as in the IEEE 754 format. For example, the result of a summation is infinity if the addition of the most significant bits of two BigInts produces a carry. Similarly, in multiplications, the resulting BigInt is set to infinity if the result is wider than the BigInt. A BigInt of all 0s is the number 0.

B. *Applicability to Network Operations*

BigInts are good candidates for network-wide operations, such as MPI reductions where communication follows a tree-like fashion and every node sums the operands it receives and then transmits the partial sum (a single number) to the next level of the tree to produce a single result [32], [33], [34]. That is because the additional bits do not have a noticeable effect in communication delay for a small number of operands, as we show in Section V-D.

Past work has shown that performing computation in the NICs during reduction operations increases both scalability and consistency with speedups of up to 121% [20]. However, this has only been applied to double-precision variables because of the complex data structures of libraries with arbitrary precision and the limited programmable logic in terms of both area and clock frequency in modern NICs [20]. For instance, Elan3 in Quadrics QsNet provides a user-programmable, multi-threaded, 32-bit, 100MHz RISC-based processor with 64 MB local SDRAM, originally targeted for communication protocol modifications [35]. This processor is an order of magnitude slower than the multi-GHz node processors. Similar work which implemented the computation logic in software drivers in a Myrinet network [18] showed that NIC-based reductions can be preferable to host-based (serial) reductions with as few as eight nodes [19].

The only way to offer high-speed computation in NICs without BigInts is to add dedicated high-speed floating-point computation hardware to NICs. The power and area consumed by a fully functional double-precision adder is substantial. In addition, including a full floating point into the NIC increases design complexity to a point that drastically increases design risk for a very specialized function. An example of the simplest double-precision FPU from the Tensilica design library requires on the order of 150,000 gates [36]. Even though FPU units support multiplication as well as addition, and therefore would be partially unused in a distributed summation, floating point multiplication is less complicated to implement than addition [37]. Verification is non-trivial for even a circuit of this size and complexity, and this implementation is among the simplest available. For future Infiniband 4x EDR interface theoretical peak bandwidth, operands would need to be summed at a rate of 100 Gigabits/sec. This would require a 32-bit carry-lookahead adder operating at 0.6 GHz (very modest clock rate). Operating as a ripple-carry, this would require 66 cycles to graduate a result, and consume about 380 gates for the adder (a negligible amount of area and power) [38].

For reduction operations with both local and network-wide computations, local computations can either use sorting or recursion techniques, or use wide fixed-point representations with dedicated hardware support. Alternatively, local computations can be performed in the NICs using the BigInt format without using the local processor. However, because this option would transfer all local operands to the NIC instead of just the partial sum, it stresses the processor–NIC interconnect. Network-wide computations are performed in the NIC with BigInts, because sorting or recursion techniques would produce an excessive amount of system-wide data movement. This way, there is no precision loss in both local and distributed operations. Local computations can be performed in advance of the arrival of packets with partial sums such that when the reduction operation function call is made, the result (a single partial sum number) can be stored in the NIC in anticipation of other partial sums (operands).

IV. Methodology

We evaluate the impact of BigInts on computation precision and performance of summations and MPI reduction operations [39] with millions to billions of operands. We first evaluate the numerical behavior and error accumulation of techniques applicable to network-wide operations, namely doubles, long doubles, DDs and BigInts, using serial test cases on large numbers of operands. Next, we evaluate the performance of *MPI reduce()* operations on payload sizes required for these different extended precision implementations. Lastly, we estimate the impact of embedding BigInt summations into the NIC for future interconnect designs.

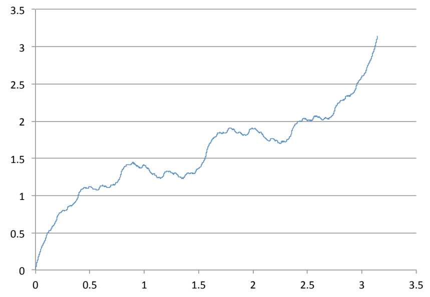


Figure 4: One benchmark is calculating the arc length of this highly irregular function (g(x)).

Numerical stability calculations are done in a single processor as they are designed entirely to understand the numerical behavior of different approaches. Each trial uses exclusively operands and operations of the appropriate precision. In our architecture, long double-precision variables are 80 bits long, where 1 bit is the sign, 15 bits are for the exponent, and 64 bits for the mantissa [40]. We report the error of each precision type against BigInt as a percentage. We also perform a bitwise comparison of the mantissa of each precision type against BigInt and report the first bit that differs. Due to the different representation with two double-precision variables that DDs use, we do not report bit-wise mantissa comparison results for DDs. Finally, we perform a comparison of decimal digits of each variable in decimal form and report the position of the first decimal digit that differs. For example, the numbers 2.54 and 2.58 differ in the second decimal digit. For both decimal and bit comparisons, a value of minus one means that there is no difference, whereas a value of 0 means that the integer parts of the numbers differ in the decimal comparison.

Initially, we perform a composite summation of numerous equal small numbers to a large number. We vary the large and small numbers as well as the number of operands to illustrate precision loss as a function of summation size and operand values. We then show results for a *funarc* calculation that attempts to calculate the arc length of the irregular function $g(x) = x + \sum_{0 \leq k \leq 10} 2^{-k} \sin(2^k x)$, over the interval $(0, \pi)$. The task here is to sum $\sqrt{h^2 + (g(x_k + h) - g(h))^2}$ for $x_k \in (0, \pi)$ divided into n subintervals, where $n = 10000000$, so that $h = \pi/10000000$ and $x_k = kh$. The function is shown in Figure 4. This function highly irregular, so that the arc length calculation will sum many highly varying quantities. If the limit on the summation were infinity instead of 10, the resulting function, while continuous and innocuous-looking, is in fact non-rectifiable—it does not have a finite arc length. In any event, it was chosen as a useful example for demonstrating round-off error when adding a large number of varying terms. Finally, to clearly show the effect of the limited mantissa bits, we use the geometric series:

$$\sum_{i=0}^{k} 2^{-i} = \frac{1 \cdot (1 - 2^{-k})}{1 - \frac{1}{2}} = 2 \cdot (1 - 2^{-k})$$

For performance results, we conduct communication and summation evaluations. To evaluate communication delay, we conduct 50000 MPI reduction operations for each precision type in NERSC's Hopper, which is a Cray XE6 cluster [41]. Because of the small payload of packets, we configure the NICs to use the eager message path to the processor [42] to avoid the latency of block transfers.

V. Evaluation

A. Composite Summation

The precision loss in the first experiment is illustrated in Figure 5. Since each summation produces an error and the small numbers (10^{-8}) we add to the large number (10^{8}) are equal, the error grows linearly with the number of operands. Also, the errors introduced by each variable type differ in orders of magnitude compared to BigInt, because doubles, long doubles and DDs have different levels of precision. DD variables introduce approximately 10^{-16} less error than doubles. For better illustration, Figure 6 (left and center) compares bits and decimal digits with the BigInt result. As shown, double and long double variables have a large error, whereas the error for DDs is still noticeable.

Figure 6 (right) shows results for a summation where the large number (10^{12}) is large enough such that double- and long double-precision variables drop the small numbers entirely. For those cases, summation is ineffective because the result equals the large number. As shown, the first mantissa bit of difference for doubles compared to BigInts is nearly the total bits in the mantissa of a double-precision variable (53), which shows that the summation affects bits that are beyond what the double-precision variable can hold. In contrast, BigInt contains all such bits. Compared to adding to a large number of 10^{-8}, the precision for DDs is also lower.

B. Computing Arc Length of an Irregular Function

To illustrate the precision loss in a realistic highly irregular mathematical computation, we use the *funarc* code on $g(x)$, described in Section IV. In this experiment, the numbers are close enough such that operands are not fully discarded. The precision loss does not increase linearly with the number of operands because the function is highly variable. Figure 7 illustrates the percentage error compared to BigInt, and Figure 8 shows the bit and decimal digit comparisons. As shown, DDs still introduce noticeable error.

C. Geometric Series

The results for the geometric series are shown in Figure 9. Even though for i infinite the geometric series converges to the number 2, for natural values of i it should converge to less than 2. However, due to the limited number of mantissa bits, double variables report equal to 2 for $i > 53$ and long doubles for $i > 64$. DDs are accurate until $i > 106$. Beyond that, the second double-precision variable, which denotes the error contained in the first double-precision variable in DDs [14], cannot accurately capture the accumulated error because the error exceeds its 53 available mantissa bits. In contrast, BigInts are accurate until $i > 1024$. If $i \leq 1024$, BigInt contains an array of consecutive asserted bits with the most significant in position 1024 and the least significant in position $1024 - i$.

D. Performance Evaluation

To evaluate the performance impact of BigInts in distributed summations we use MPI reductions and measure communication time using MPI constructs. MPI reductions are typically used for summations performed on operands distributed across a large-scale system [32], [34]. In such operations, communication progresses in a tree-like fashion where nodes in each level of the tree sum the operands they receive with their local operands, and pass the partial sum (a single number) to the next level [33].

Figure 10 shows the time spent in communication. BigInt increases communication time by approximately 35% compared to doubles, but only 2%-14% compared to DDs. Even though BigInts contain 2101 bits and therefore are 16$\times$ larger than 128-bit DDs, for such small sizes communication is latency-bound instead of throughput-bound. Therefore, BigInt does not increase communication time significantly because the additional bits lead to a minor increase since packet headers and fixed (zero-load) delays in the network and NICs dominate for small transfers. Our results are confirmed by past work [21] which showed that the communication delay for messages with payloads containing double-word, quad-word, and double quad-word variables is identical and approximately 12 μs for up to 512 bytes, which is a larger payload size than BigInts (256 bytes).

We then estimate computation time. In modern Intel processors, double-precision additions performed on dedicated FPUs require 5 cycles [27]. Even though the peak rate for FPUs is 2 flops/cycle, summation throughput is limited by the FPU pipeline latency. Increased-precision representations are considerably slower. For example, as discussed in Section II, DD addition requires up to 20 operations. Arbitrary-precision libraries use more complex representations and

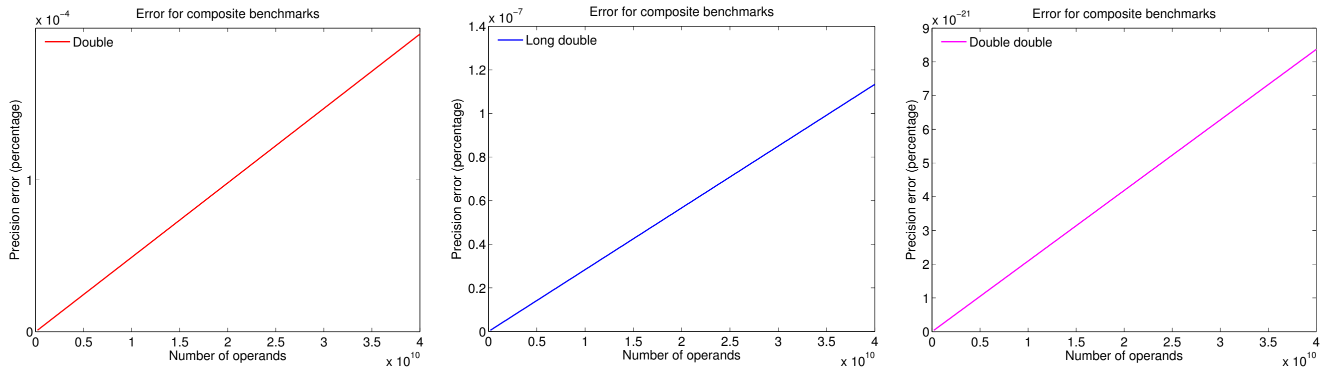


Figure 5: Adding operands with a value of 10^{-8} to a variable with the value of 10^{8}. BigInt equals the analytical result.

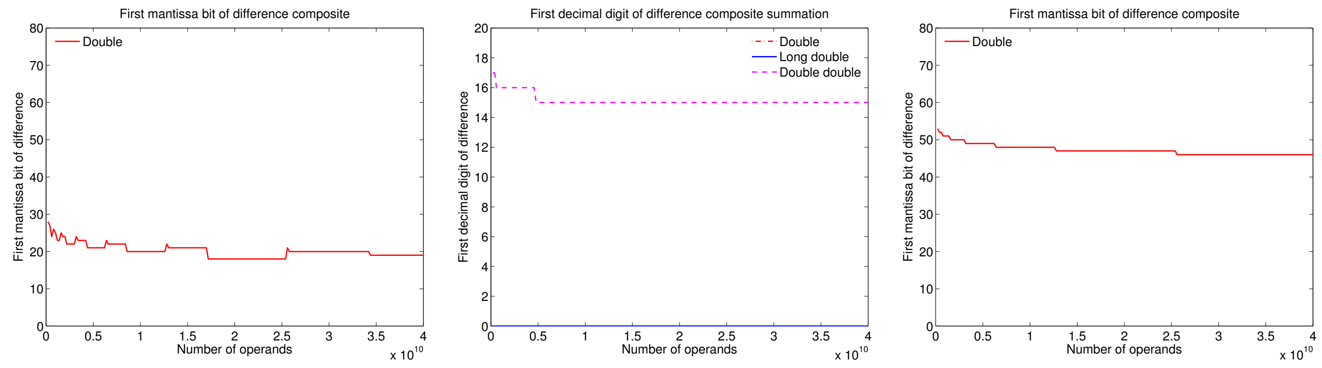


Figure 6: Adding operands with a value of 10^{-8} to a variable with the value of 10^{8} (left and center) or 10^{12} (right). In the center Figure, the lines for double and long double overlap on zero because the integer part also differs.

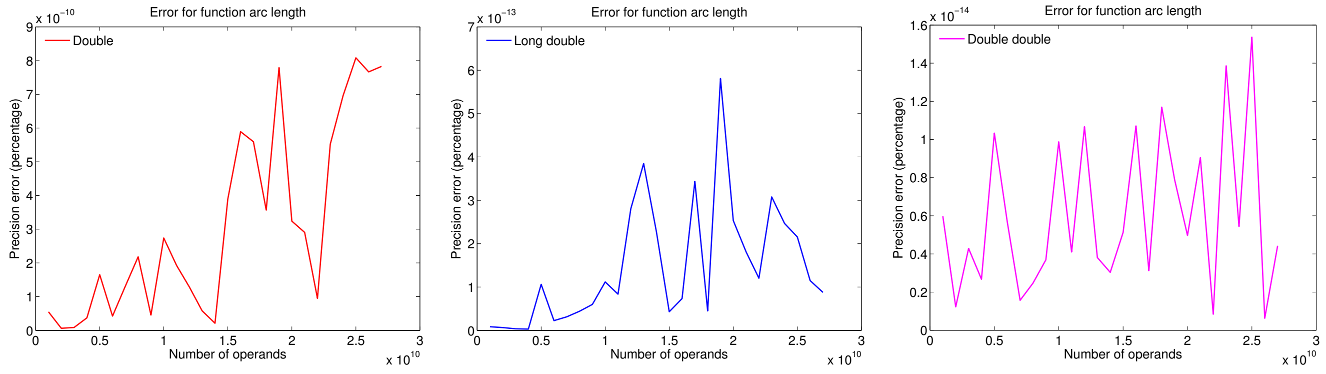
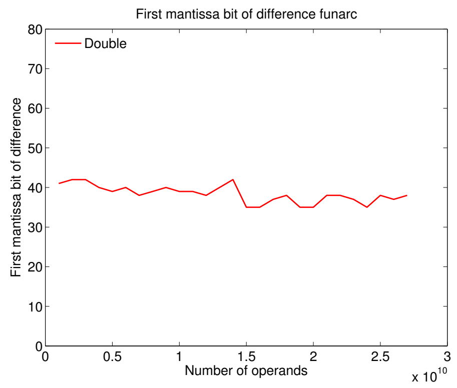


Figure 7: Precision error compared to BigInt for computing the arch length of g(x) with a varying number of operands.

therefore are even slower. BigInts can perform these operations at a rate that is comparable to DDs (20 cycles) only with an 106-bit integer adder, which can be easily integrated in modern NICs [20]. Computing in NICs avoids potentially large context switching or wake up delays for local processors, as well as data transfer latencies between the NIC and the processor that are typically significantly longer than the few cycles to perform integer addition. Context switching requires μs [43], while waking up processors from deep sleep may require seconds [44], [45]. So the handful additional cycles required for extended precision and BigInts are negligible compared to the network time. As discussed in Section III-B, the complexity of integer adders is an order of magnitude less than FPUs. Therefore, compared to FPUs, BigInt summations are feasible to perform at line rate using existing technology and consuming an order of magnitude fewer gates, using only integer adder components in a pipelined ripple-carry fashion. Therefore, BigInts enable



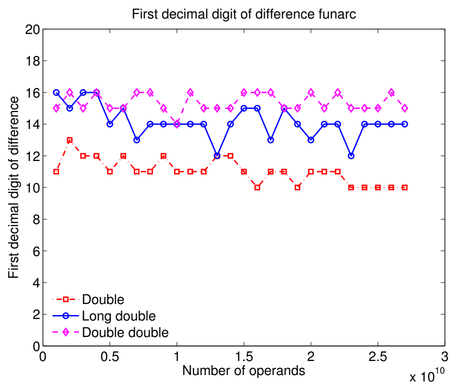


Figure 8: Bit and decimal digit comparison for computing the arch length of g(x) with a varying number of operands.

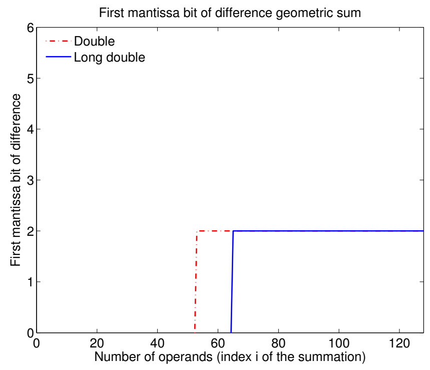


Figure 9: Bit comparison for the geometric series. DDs are not shown due to their different internal representation.

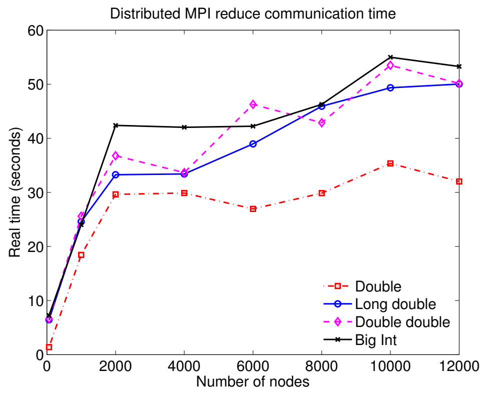


Figure 10: Communication time for an MPI reduce operation.

reduction operations to complete in comparable time as operations with double-precision variables, while providing no precision loss in the same number space with exact encoding for every real number in that space.

VI. Conclusion

This paper makes the case that using a very wide integer, which we call BigInt, is well-suited for distributed operations. BigInts encode the same number space as double-precision variables but have a simple enough format to perform operations such as additions in comparable time as double-precision variables with dedicated FPUs, using inexpensive hardware integer adders or programmable logic found in modern NICs. BigInts enable distributed reduction operations in large-scale systems with no precision or performance loss. Computing in NICs avoids the energy and latency costs of NIC to processor communication, as well as context switching or waking up the processor. Computing in NICs is only achievable by past work with double-precision variables due to the complex internal structures and system-wide data movement introduced by increased- or arbitrary-precision software libraries and algorithms.

Disclaimer

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

References

[1] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *9th international conference on High performance computing for computational science*, ser. VECPAR'10, 2011.

[2] D. H. Bailey, "High-precision floating-point arithmetic in scientific computation," *Comp. in Science Engineering*, vol. 7, no. 3, 2005.

[3] D. H. Bailey, R. Barrio, and J. M. Borwein, "High-precision computation: Mathematical physics and dynamics," *Applied Mathematics and Computation*, vol. 218, no. 20, 2012.

[4] K. R. Ghazi, V. Lefevre, P. Theveny, and P. Zimmermann, "Why and how to use arbitrary precision," *Computing in Science and Engineering*, vol. 12, no. 3, p. 5, 2010.

[5] E. Allen, J. Burns, D. Gilliam, J. Hill, and V. Shubov, "The impact of finite precision arithmetic and sensitivity on the numerical solution of partial differential equations," *Mathematical and Computer Modelling*, vol. 35, no. 11-12, 2002.

[6] J. M. Chesneaux, S. Graillat, and F. Jézéquel, "Rounding errors," in *Wiley Encyclopedia of Computer Science and Engineering*, 2008.

[7] Y. He and C. H. Q. Ding, "Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications," in *14th international conference on Supercomputing*, ser. ICS '00, 2000.

[8] J. Demmel, I. Dumitriu, O. Holtz, and P. Koev, "Accurate and efficient expression evaluation and linear algebra," *Computing Research Reporitory*, vol. abs/0712.4027, 2007.

[9] J. M. McNamee, "A comparison of methods for accurate summation," *ACM SIGSAM Bulletin*, vol. 38, no. 1, 2004.

[10] J. Demmel and H. D. Nguyen, "Fast reproducible floating-point summation," in *21st IEEE Symposium on Computer Arithmetic*, 2013.

[11] J. Demmel, B. Diament, and G. Malajovich, "On the complexity of computing error bounds," *Foundations of Computational Mathematics*, vol. 1, no. 1, pp. 101–125, 2001.

[12] N. J. Higham, "The accuracy of floating point summation," *SIAM Journal on Scientific Computing*, vol. 14, 1993.

[13] S. Graillat and V. Ménissier-Morain, "Accurate summation, dot product and polynomial evaluation in complex floating point arithmetic," *Information and Computation*, no. 216, pp. 57–71, 2012.

[14] Y. Hida, X. Li, and D. H. Bailey, "Library for double-double and quad-double arithmetic," http://web.mit.edu/tabbott/Public/quaddouble-debian/qd-2.3.4-old/docs/qd.pdf/, 2007.

[15] S. Siegel and J. Wolff von Gudenberg, "A long accumulator like a carry-save adder," *Computing*, vol. 94, no. 2-4, pp. 203–213, 2012.

[16] A. Chervenak *et al.*, "Data placement for scientific applications in distributed environments," in *8th IEEE/ACM International Conference on Grid Computing*, ser. GRID '07, 2007.

[17] A. Vishnu, M. Koop, A. Moody, A. Mamidala, S. Narravula, and D. Panda, "Hot-spot avoidance with multi-pathing over InfiniBand: An MPI perspective," in *7th IEEE International Symposium on Cluster Computing and the Grid*, ser. CCGRID '07, 2007.

[18] N. J. Boden *et al.*, "Myrinet: A gigabit-per-second local area network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, 1995.

[19] D. Buntinas and D. K. Panda, "NIC-based reduction in Myrinet Clusters: Is it beneficial?" in *SAN-02 Workshop*, 2003.

[20] F. Petrini, A. Moody, J. Fernandez, E. Frachtenberg, and D. K. Panda, "NIC-based reduction algorithms for large-scale clusters," *International Journal on High Performance Computer Networks*, vol. 4, no. 3/4, pp. 122–136, 2006.

[21] A. Vishnu, M. ten Bruggencate, and R. Olson, "Evaluating the potential of Cray Gemini interconnect for PGAS communication runtime systems," in *19th IEEE Annual Symposium on High Performance Interconnects*, ser. HOTI '11, 2011.

[22] U. Kulisch and V. Snyder, "The exact dot product as basic tool for long interval arithmetic," *Computing*, vol. 91, no. 3, 2011.

[23] U. Kulisch, "Very fast and exact accumulation of products," *Computing*, vol. 91, no. 4, pp. 397–405, 2011.

[24] "IEEE standard for floating-point arithmetic," *ANSI/IEEE Std 754-2008*.

[25] T.-J. Kwon, J. Sondeen, and J. Draper, "Design trade-offs in floating-point unit implementation for embedded and processing-in-memory systems," in *IEEE International Symposium on Circuits and Systems*, ser. ISCAS '05, 2005.

[26] X. Hong, W. Chongyang, and Y. Jiangyu, "Analysis and research of floating-point exceptions," in *2nd International Conference on Information Science and Engineering*, ser. ICISE '10, 2010.

[27] "Intel 64 and IA-32 architectures developer's manual: Vol. 1," Intel Corporation, March 2012.

[28] D. H. Bailey, Y. Hida, X. S. Li, and O. Thompson, "ARPREC: An arbitrary precision computation package," Lawrence Berkeley National Laboratory, Tech. Rep., 2002.

[29] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5th ed., 2012.

[30] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Transanctions on Mathematical Softwware*, vol. 33, no. 2, 2007.

[31] V. A. CarreÃśo and P. S. Miner, "Specification of the ieee-854 floating-point standard in hol and pvs," 1995.

[32] H. Ritzdorf and J. Traff, "Collective operations in NEC's high-performance MPI libraries," in *International Parallel and Distributed Processing Symposium*, ser. IPDPS '06, 2006.

[33] T. Hoefler and J. Traff, "Sparse collective operations for MPI," in *29th IEEE International Symposium on Parallel Distributed Processing*, ser. IPDPS '09, 2009.

[34] T. Kielmann, R. E. H. Hofman, H. E. Bal, A. Plaat, and R. A. E. Bhoedjang, "MPI's reduction operations in clustered wide area systems," in *Message Passing Interface Developer's and User's Conference*, ser. MPIFC '99, 1999.

[35] F. Petrini, W.-c. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The quadrics network (QsNet): High-performance clustering technology," in *Proceedings of the The Ninth Symposium on High Performance Interconnects*, ser. HOTI '01, 2001, pp. 125–130.

[36] J. Krueger *et al.*, "Hardware/software co-design for energy-efficient seismic modeling," in *Conference on High Performance Computing Networking, Storage and Analysis*, 2011.

[37] G. Govindu, L. Zhuo, S. Choi, and V. Prasanna, "Analysis of high-performance floating-point arithmetic on FPGAs," in *18th IEEE International Parallel and Distributed Processing Symposium*, ser. IPDPS '04, 2004.

[38] R. H. Katz, *Contemporary logic design*. Benjamin-Cummings Publishing Co., Inc., 1993.

[39] M. P. I. Forum, "MPI: A message-passing interface standard. version 3.0," http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf/, 2012.

[40] "IEEE standard for binary floating-point arithmetic," *ANSI/IEEE Std 754-1985*.

[41] K. Antypas, "The Hopper XE6 system: Delivering high end computing to the nation's science and research community," Cray Quarterly Review, Tech. Rep., April 2011.

[42] H. Pritchard, I. Gorodetsky, and D. Buntinas, "A uGNI-based MPICH2 nemesis network module for the Cray XE," in *18th European MPI Users' Group conference on Recent advances in the message passing interface*, ser. EuroMPI'11, 2011.

[43] D. Tsafrir, "The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)," in *Experimental computer science on Experimental computer science*, ser. ECS '07. USENIX Association, 2007.

[44] S. Damaraju *et al.*, "A 22nm IA multi-CPU and GPU system-on-chip," in *59th IEEE International Solid-State Circuits Conference Digest of Technical Papers*, ser. ISSCC '12, 2012.

[45] L. Case, "Inside Intel's Haswell CPU: Better performance, all-day battery," http://www.pcworld.com/article/262241/inside_intels_haswell_cpu_better_performance_all_day_battery.html/, 2012.